Which end of the data egg gets broken first—big or little? Do you start with the MSB or the LSB? For the proposed IEEE 896 bus, the most practical approach may be the little-endian one.

Data Format and Bus Compatibility in Multiprocessors

Hubert Kirrmann
Brown Boveri Research Center

Several bus standards are being developed by the IEEE, such as the proposed IEEE 802 standard for local-area networks. Other standardization efforts by organizations such as the ISO and the IEC apply to industrial buses and long-haul networks. Yet other standards aim at closely coupled systems. The P896 backplane bus,[1] a proposal being developed by the IEEE Computer Society, the European Workshop on Industrial Computing Systems, and the Institution of Electrical Engineers, has been designed to be the

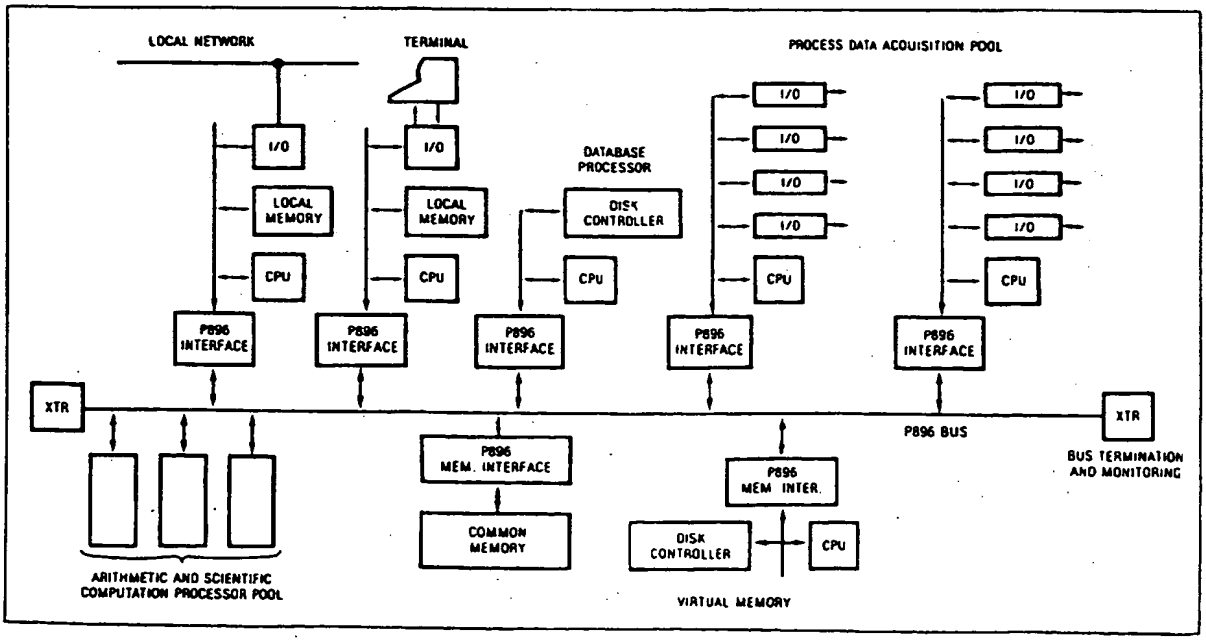

Figure 1. A nonhomogeneous tightly coupled multiprocessor.

0272-1732/85/0800-0032$01.00 © 1985 IEEE

backbone of a tightly coupled multiprocessor system in which different boards communicate through a common memory (Figure 1).

Standards should guarantee compatibility between products from different manufacturers and of different technical conception. Computing systems interconnected by standard buses are often nonhomogeneous, and the standard has the difficult task of guaranteeing communication between building blocks having different speeds, signal protocols, and data formats.

The lower protocol layers, which ensure mechanical, electrical, and timing compatibility, rule *how* information is transmitted, not *what* information is transmitted. Beyond this physical compatibility, the modules must agree on a common format for data transmission and storage. Such a convention requires a much broader standardization than the physical layers, since it directly concerns the instruction set and the architecture of the processors.

Until now, the manufacturers of processors have not agreed on any common data representation, with the result that it is not possible for a processor to interpret data in memory without knowing its source. An automatic translation within the bus interface is usually not feasible, since the interface would need an impractical amount of knowledge about what the processor is intending to do.

The problem gets worse when software layers are involved. It is reasonable to assume that programs that run on different processor types are compiled independently. These programs must, however, communicate with each other by means of common data structures such as mailboxes and ports. Unfortunately, compilers have little respect or knowledge of other compilers' conventions, so most of the time compatibility has to be achieved by rather primitive and inefficient mechanisms. The data compatibility problem is present each time data are exchanged between machines of different type or even of different serial number. It appears in multiprocessor computers, in networks, and each time a tape or a floppy disk written on one type of machine must be read on another.

In this article, I analyze the problem of data representation and show to what extent it can be solved at the bus interface level. I then consider how future 32-bit processors should be interfaced, and I give the rationale for the choice of the bus data format in the EDISG* proposal for the IEEE P896 Future Bus standard.

Data format compatibility

Each time data are transferred between computers of different types, a common data format must be employed. This is not easy to achieve, since the user normally has no control over data formats, especially if she or he programs in a high-level language. Suppose that a

*EDISG, the European Distributed Intelligence Study Group, is committee TC 10 of EWICS, the European Workshop on Industrial Computing Systems. EWICS is supported by the European Community.

Little-endians vs. big-endians

In Jonathan Swift's *Gulliver's Travels*, Gulliver, the author's hero, is shipwrecked and washed ashore on Lilliput, whose six-inch-tall inhabitants are required by law to break their eggs only at the little ends. Of course, all those citizens who habitually break their eggs at the big ends are angered by the proclamation. Civil war breaks out between the little-endians and the big-endians, resulting in the big-endians taking refuge on a nearby island, the kingdom of Blefuscu. The controversy is ethically and politically important for the Lilliputians—Swift has 11,000 Lilliputian rebels die over the egg question.

Swift was satirizing the causes of the devastating religious wars of his day. His point was that warring over matters of religious conviction is just as absurd as warring over egg-breaking—that everyone should follow his own preferred way.

In his October 1981 article in *Computer*, "On Holy Wars and a Plea for Peace," Danny Cohen applied Swift's terms to the debate over data format in the microcomputer world. Cohen asserted that the difference between sending information with the little or big end first is indeed trivial, but that *agreement* on a single way of sending is not trivial at all. To avoid anarchy in the microcomputer kingdom, everyone must break the data egg on the same end. Which end is unimportant, agreement on it is. Cohen suggested a coin toss.

Here, Hubert Kirrmann investigates the problems encountered in interfacing both little-endian and big-endian devices to a standard microcomputer system bus. The bus is the proposed IEEE 896 backplane, and Kirrmann recommends that it be little-endian. He favors that approach not as a matter of technical conscience but as one of technical practicality. Ease of interfacing, low cost, and coming developments in the microcomputer kingdom favor the little end. Hence, the coin toss may *not* be the solution of choice for the citizens of the 896 province.

processor is sampling data and that it sends them for processing to another processor. The sending program has been written in Pascal and defines the data to be sent as being of type "exchange":

```
TYPE exchange = RECORD OF
                  date: (daytime = ARRAY[1:12]
                  OF CHAR);
                  IO_channel: (channel = 0..255);
                  sample: (measure = REAL)
                  END;
```

Even if the receiver program has also been programmed in Pascal and also defines the data as being of type "exchange," it is unlikely that the data will be correctly interpreted if the computers are different or even if the computers are the same but are running two different compiler versions. So an underlying common data format must be established to allow programs to exchange data in a consistent way. At the end of this article, a proposal for a standardized data format is given. Ideally, the agreement on the data format should be enforced by the compilers, not by the processor or by the user. As this is not the case today, the user must have a knowledge of the underlying data structure at least for all routines that communicate with the outside world. So we will first review the data formats used by most of today's processors.

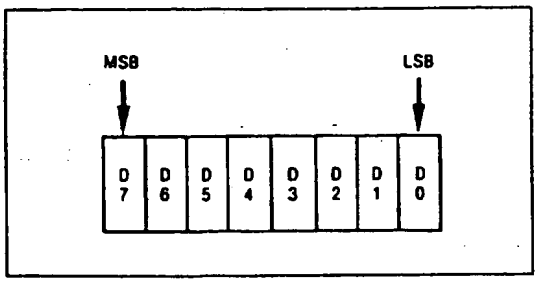

Figure 2. One byte represented in little-endian notation.

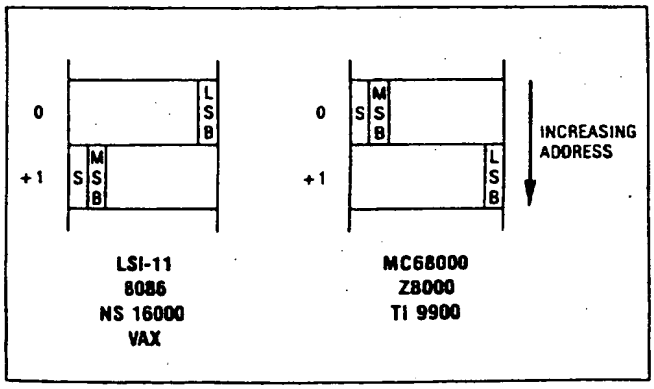

Figure 3. 16-bit integer formats.

Processor fixed-data types

In the following, we shall compare the data formats used by different processors. We will consider only the data structures recognized at the instruction set level by the hardware. We will distinguish between those data types that are *public* and that can be exchanged with other processors, like integers, reals, and pointers, and those that are *private* and are unique to a processor, like the instruction format and descriptors and call frames.

Software-dictated data structures, like sets, files or records, and system configuration tables, are not considered, although these data structures are already being cast in silicon today. We will come back later to the case of the software data structure.

We shall use "byte" to mean an 8-bit unit of information, "word" a 16-bit unit, "triplet" a 24-bit unit, and "quad" a 32-bit unit.

Byte. The only data format every microprocessor today agrees upon is that memory consists of a linear array of bytes. This doesn't apply to big mainframes, which consider memory to be an array of words, quads, or even rare entities like 24-bit or 36-bit chunks, or to some minicomputers like the PDP-8, which consider the memory to be an array of 12-bit units.

Within a byte, we should give each bit a number. Here is where the trouble begins: Should we name D0 the most significant bit (MSB) or the least significant bit (LSB)? Both conventions are meaningful. Most microprocessors today name D0 the LSB, but the TI 9900 and the PDP-8, as well as most big mainframes, have the reverse convention and name D0 the MSB or sign bit. The reasons (or rather unreasons) for this difference are well explained in an article by Cohen.[2] We will follow Cohen's notation and refer to the first convention as "little-endian" (LE) and the second convention as "big-endian" (BE).

We will observe the following definitions:

- In a little-endian format, the least significant digit or bit has the lowest number and is stored at the lowest address.
- In a big-endian format, the most significant digit or bit has the lowest number and is stored at the lowest address.

Interestingly enough, the big-endian/little-endian controversy takes place within the same company (DEC and TI have products that employ opposing conventions) and even within the same processor, as we will see. The concept of little-endian or big-endian is thus not a property of a processor. A processor is LE or BE only with respect to certain data types.

Since we cannot remain neutral, we adopt for counting bits the LE notation, which is the most common for microprocessors (Figure 2). In order to have a reference point, we shall try to remain little-endian in this article.

16-bit integer (word). The LSI-11,[3] 8086,[4] NS 16000,[5] iAPX 432[6] store the most significant part of a word (MSP) with the sign bit at the higher address and are, ac-

cording to our definition, little-endians for words. The MC68000,[7] the Z8000,[8] and the TI 9900[9] store the MSP at the lowest address and are therefore big-endians for words. We see already that the MC68000 and Z8000 are inconsistent, since they are little-endians for counting bits but big-endians for counting bytes (Figure 3).

32-bit integers. Here again, the data representations are quite different. The 432, 8086/8087, NS 16000, and VAX[10] follow the little-endian philosophy, while the 68000 goes the big-endian way, and DEC's LSI-11 takes a middle way—little-endian for words but big-endian for quads (Figure 4).

Address. Processors normally consider an address to be a 16-bit or 32-bit integer and use the same representation for it as for an integer of the same size. Memory addresses are public data only within a tightly coupled multiprocessor. It makes no sense to transmit a pointer to data in a network; however, in a network, a device name can be considered an address. The address format must therefore also be standardized in a tightly coupled multiprocessor. It is relatively easy to enforce a common address by mapping, as long as all modules agree that the address space is divided into bytes. Here, the little-endian notation has the nice effect that it does not require that all lines be renamed as address size is increased.

Floating-point formats and BCD. Although a proposal for a floating-point format has been submitted to the IEEE,[11] there is no agreement upon how the 32- or 64-bit string representing the number should be stored, so everybody does it his own way (Figure 5). The only consistent way is the Intel way, which stores the numbers in the little-endian format. DEC[2,10] uses a mix between LE and BE.

For BCD, as above, every manufacturer has his own format (Figure 6).

Compound structures. Compound structures are arrays, records, files, or some combination thereof. There are only a few hardware-defined compound structures, the most common being the array of characters, or string.[10] A BCD representation of a number is not an array, as long as the structure is treated as a whole by the machine. Fortunately, all processors agree that in an array structure, the first element of the array is stored at the lowest address.

Some processors require character strings to have a length field, others require them to have a trailing delimiter. For these cases, the structure can be viewed as a record consisting of two elements, an array of characters, and a delimiter or length field.

Urgency of standardization. There is no common data representation among processors. As processor complexity increases, more data types are put in silicon and the compatibility problem increases. The 8080 has only two hardware-defined data types (bytes and words), whereas the VAX has about 10. And the coprocessors now being put on the market are further increasing the number of hardware-defined data formats, and are doing so rapidly. Hence, adoption of a standard data representation is an urgent matter.

Data formats on a parallel bus between a processor and memory

The data format in memory does not depend solely on the processor. Every bus standard today imposes—unnecessarily as we will see—a data format for transmis-

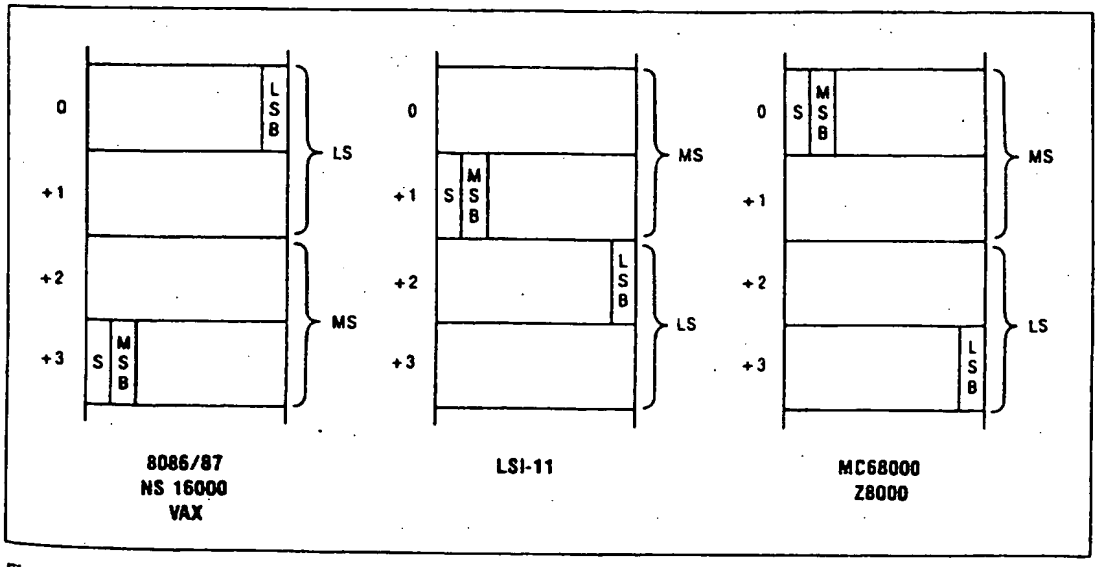

Figure 4. 32-bit integer formats.

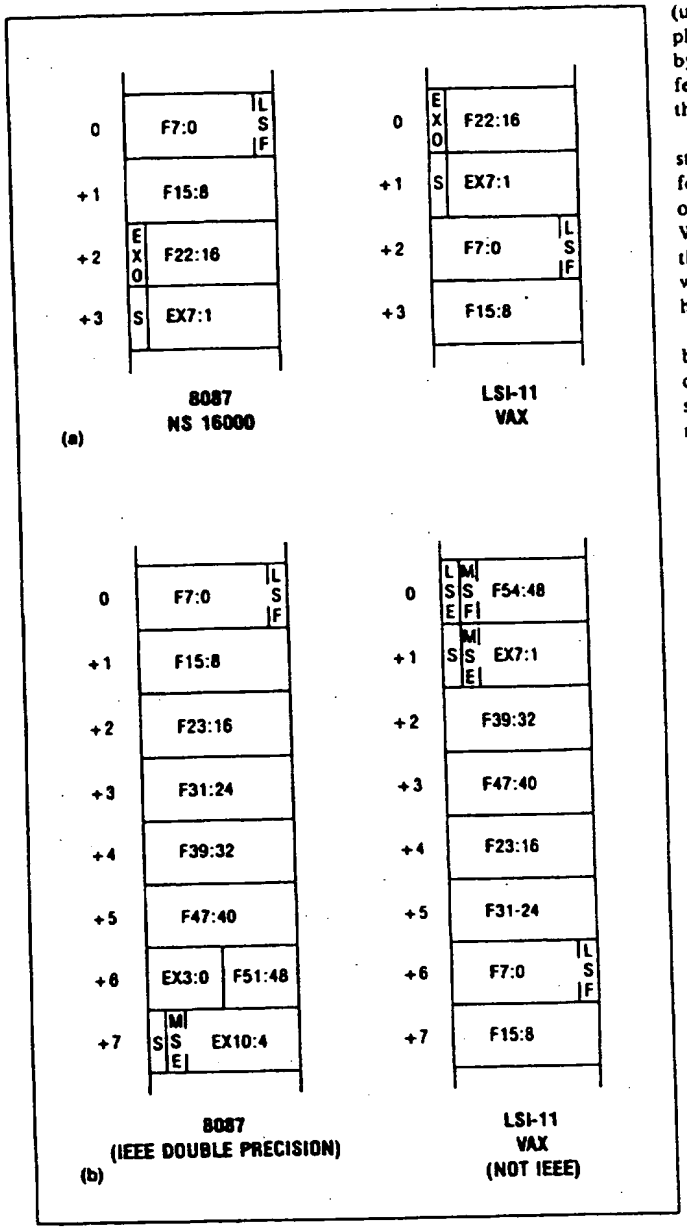

Figure 5. Floating-point formats—for 32-bit numbers (a) and 64-bit numbers (b).

sion and storage. The data format on a parallel bus depends on three factors: data size, memory alignment, and justification.

Data size. The bus width is the largest chunk of data that the bus can transmit in parallel in one operation (e.g., 32 bits), and the bus data unit is the smallest chunk of data that the bus can transmit at a time as a unit (usually one 8-bit byte). In a 16-bit-wide bus, for example, a transfer can consist of a 16-bit word or of an 8-bit byte. In a 32-bit-wide bus, information can be transferred by bytes, words, triplets, or quads. In both cases, the bus data unit is the byte.

The bus width and data unit size dictate the memory structure. A memory has exactly one independent bank for each bus data unit, e.g., one bank per byte. Any other arrangement would require a READ/MODIFY/WRITE operation for each WRITE that is smaller than the bank size. Just think how a nibble (4 bits) should be written into a half-byte without disturbing the other half.

A memory should always have the same width as the bus itself, since the speed gain of a wide bus would otherwise be offset by having to store data in consecutive locations in the same chip (which requires two memory cycles).

Alignment. Alignment is an additional restriction imposed by a processor on the data representation in memory in order to simplify the interface between the processor, the bus, and the memory.

In a byte-aligned memory, any data item (byte, word, or quad) can begin at any byte boundary, i.e., at any address. In a word-aligned memory, a byte can begin at any address but a word or a quad is constrained to begin at an even address (word address). In a fully-aligned memory, a data item can only be stored at a memory address which is an integer multiple of the item's length.

Alignment is related to the width of the bus. If the bus is 8 bits wide, there is no reason to align data. All 8-bit processors are byte-aligned. Their words can begin at any byte address. All 16-bit processors are word-aligned on the bus in order to simplify memory addressing (Figure 7).

If a word is aligned on an odd address, the memory chip address is not the same in both banks. This complicates memory control, especially if memory-board boundaries are crossed (i.e., if the higher byte is on one board and the lower on the other).

All existing 16-bit processors are word-aligned on the bus, although the NS 16000 and 8086 claim that any data item can be placed at any byte boundary. This is only true at the instruction set level, since these processors execute in reality two FETCH/STOREs when a word begins at an odd address. So instead of more complicated memory logic, a double number of memory cycles is used for READs as well as for WRITEs each time an odd-aligned word is transmitted. The speed penalty of this operation can be somewhat compensated for by instruction prefetching. We will refer to this kind of processor as "pseudo-byte-aligned."

Although all 8-bit microprocessors are by nature byte-aligned, they should respect word-alignment when writing into memory, or a word-aligned 16-bit processor will not be able to read them. The same is true for 16-bit processors in a 32-bit world.

Data in a 32-bit memory can be aligned on byte or word boundaries (Figure 8). Thirty-two-bit buses are always fully aligned, since any other arrangement would require a complicated byte routing system such as a 4

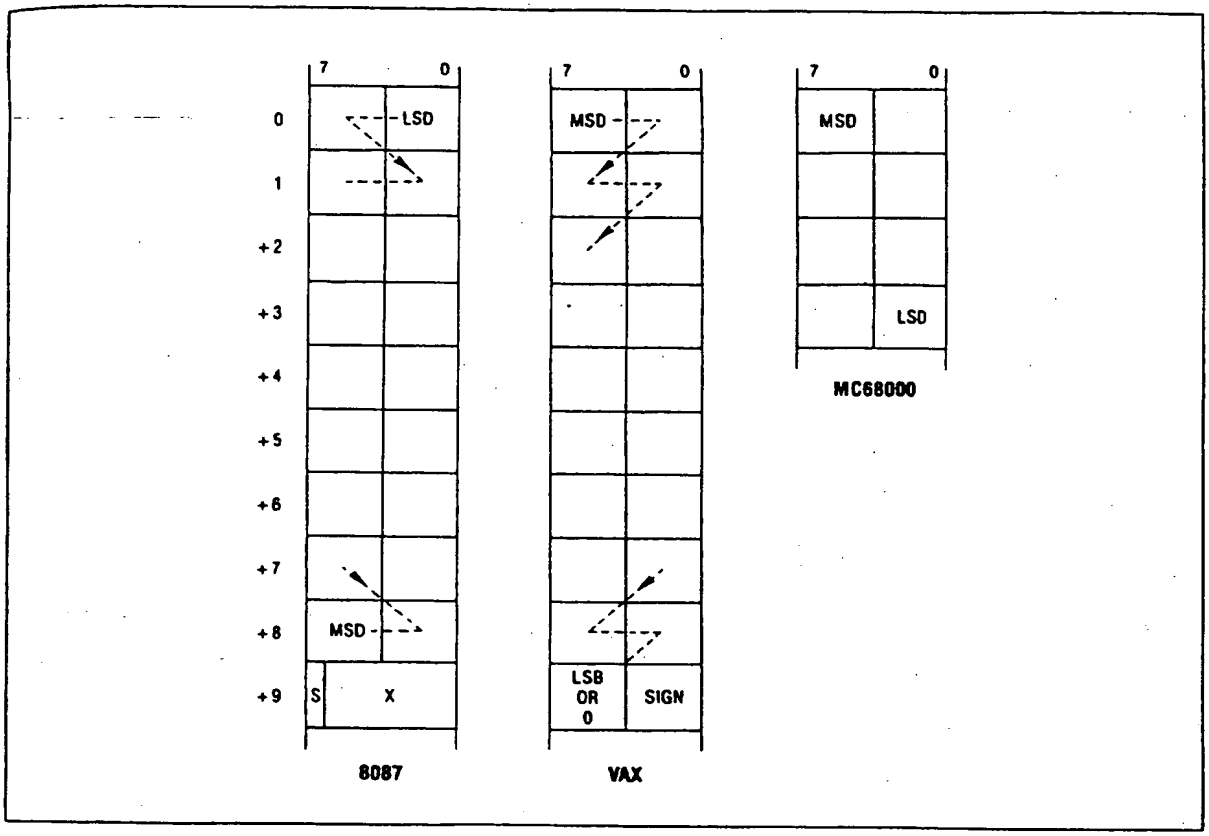

Figure 6. BCD string formats.

× 4 crossbar matrix or a 16-byte switch. Some 32-bit processors like the VAX are byte-aligned, and so they require two bus cycles for every data item that crosses a quad boundary. Interestingly, this occurs three-fourths of the time if the quads are stored randomly.

Full alignment complicates assemblers and compilers and results in some loss of storage, but reduces logic complexity in both the processor and the memory and speeds up execution. Furthermore, the programmer need not be aware of memory-board boundary crossings.

Straight (nonjustified) bus. In a straight, or nonjustified bus, the bus lanes are direct extensions of the memory banks. In a 16-bit bus, the lanes are termed "odd" and "even." In a 32-bit bus they are named 0, 1, 2, and 3, with bank 0 being accessed when the two lower bits of the address are 0. The data path that comes out of all processor chips today is straight. Straight buses include the LSI-11 bus, which has a little-endian data format, and the 16-bit Versabus, which is big-endian. The P896 bus is also straight and it has an uncommitted little-endian format.

Some processors such as the MC68000 make the lane assignment explicit by issuing one control signal per byte lane, e.g., upper data strobe, lower data strobe. Other

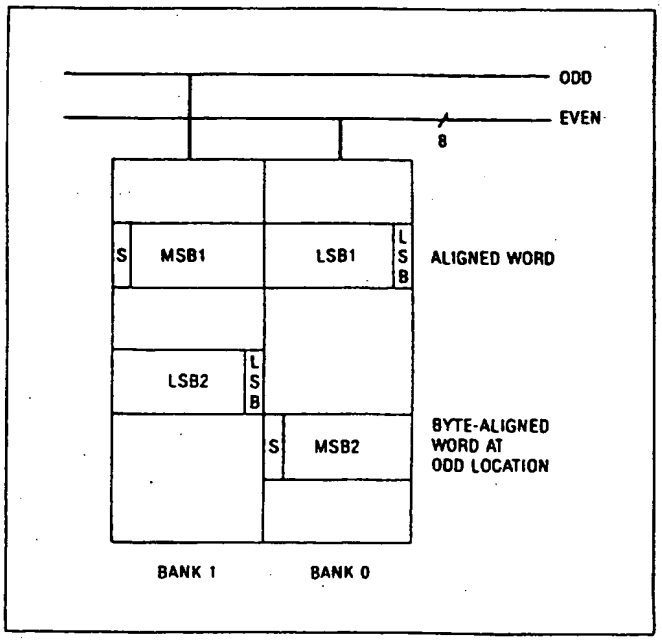

Figure 7. Storage of a 16-bit word in a byte-aligned memory (little-endian representation).

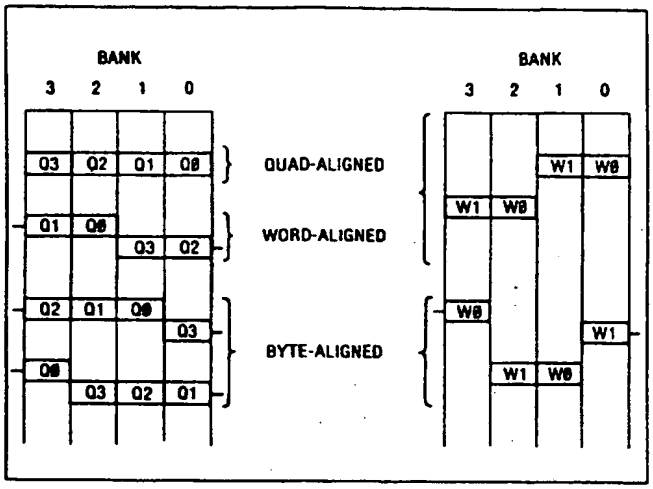

Figure 8. Quad and word storage in a 32-bit memory, with effects of alignment shown. Bytes can be stored anywhere.

processors code this information using the lowest address bit(s) and size (byte/word) information, e.g., A0 and byte/word (LSI-11, 8086, and NS 16000), which select the correct lane. (The A0/size solution saves one line if address and data are multiplexed.) Figure 9 shows how a big-endian and a little-endian processor are connected to a straight bus.

Since straight bus lanes are just extensions of memory banks, a processor has full control over storage in memory. Hence, the bus should not prejudice the data format used. But, in reality, the bus of Figure 9 is a hidden little-endian. For bytes to be stored at the correct address, the MSB line of the big-endian processor must be connected to the B7 bus line, and the LSB line to the B8 line. This could puzzle many a designer, since we have named the bus lines of Figure 9 using a little-endian notation. If the bits of a word were counted the big-endian way, as in the TI 9900, D0 would be the MSB and it would be connected to B15, and D1 would be connected to B14, and so on. What adds to the confusion is that most big-endian processors such as the Z8000 use the little-endian convention when counting the bits of a word or an address. In this case, D15 of the processor must be connected to B7, and D0 to B8.

So, just by naming the bus lines, we have favored a particular data format in memory. For instance, DEC's LSI-11 bus is straight, but just by specifying that

BDAL0 (address bit zero)
= 1 selects the high byte = BDAL<15:08>,

it declares itself a little-endian. The bus is overspecified and it needlessly rules out processors of the opposite type. Fortunately, this is only a problem of naming. The designer can avoid the pitfall if he makes sure that any general-purpose nonjustified bus is not named from D0 to D15, but has an independent notation for each byte lane.

Bus control signals

The table below shows which signals the different processor types use to control the bus. These signals select the high or low byte of the bus. There exist two variants, the A0/byte signal, which is used by almost every processor, and the UDS/LDS (upper data strobe, lower data strobe) signal, which is used by the MC68000.

PROCESSORS	BYTE SELECT	ALIGNMENT	WORD FORMAT
MC68000	UDS, LDS	WORD-ALIGNED	BE
Z8000	A0 AND B/W	WORD-ALIGNED	BE
TI 9900	NONE	WORD-ALIGNED	BE
8086	A0 AND BHE	PSEUDO-BYTE-ALIGNED	LE
NS 16032	A0 AND HBE	PSEUDO-BYTE-ALIGNED	LE
LSI-11	A0 AND WTBT	WORD-ALIGNED	LE
VAX	MASK <3:0>	PSEUDO-BYTE-ALIGNED	LE
BUSES	**BYTE SELECT**	**JUSTIFICATION**	**WORD FORMAT**
S-100	A0 AND sXTRQ	BYTE-JUSTIFIED	BE
MULTIBUS	ADR0 AND BHEN	BYTE-JUSTIFIED	LE
Q-BUS	AD0 AND WTBT	NOT JUSTIFIED	—
SBI	MASK <3:0>	NOT JUSTIFIED	—
VERSABUS, VME BUS	A01 AND LWORD, DS1 AND DS0	WORD-JUSTIFIED	BE
P896, DRAFT 5.2	COMMANDS <3:0>	SUBSET: WORD-JUSTIFIED; FULL WIDTH: NOT JUSTIFIED	LE —

KEY:
LE = LITTLE-ENDIAN
BE = BIG-ENDIAN
UDS = UPPER DATA STROBE
LDS = LOWER DATA STROBE
BHE = BYTE HIGH ENABLE
WTBT = WRITE BYTE
HBE = HIGH BYTE ENABLE
A0 = ADDRESS BIT ZERO

What happens if one tries to store the words correctly by inverting the byte lane connections of one of the processors (Figure 10)? (Forget about the byte switch BS for the moment.) Alas, this method only works with fully aligned words; an odd-aligned word will be stored differently by both processors—in the correct lane, but at the wrong address. At best, this method can be used to integrate aligned processors such as the MC68000 into a little-endian system. If the processor is fully aligned, however, one can even store bytes at the correct address. This can be done by routing a single byte to the correct lane with the help of the byte switch shown in Figure 10. To do so, one must rely on the byte/word signal that most processors give to know when to swap the byte.

Unfortunately, this can lead to trouble. First of all, not all processors issue byte/word information. The TI 9900, for instance, has no such line; it systematically accesses its memory as an array of words. To write a byte, it always does a READ/MODIFY/WRITE. The TI 9900 will store its words the big-endian way no matter what the bus format. Other processors like the LSI-11/2 do have a byte/word line, but the information on it is simply ignored. Like the TI 9900, this processor always accesses the memory by words and does a READ/MODIFY/WRITE operation on the interesting half. Even worse, some processors like the LSI-11/23 issue the byte/word indication, but only for WRITEs. They always read a word and internally select the accessed byte.

Pseudo-byte-aligned processors like the 8086 issue a byte/word indication, but it cannot be relied on. "Byte" does not mean that a single byte has been transmitted—the interface cannot distinguish the writing of two halves of a word from the writing of two separate bytes. These processors will still store the odd-aligned words in a little-endian format and will do so independently of the bus format.

So the circuit of Figure 10 can only be used in some restricted cases; it is not recommended for general use. The designer should stick to the rule that the bytes should be at the correct place, and he should take into account that the words can be inconsistent.

Justified bus. Justification, as in typography, means that data which are not as wide as the bus are bound to the left or the right of the path. A bus is byte-justified if a single byte always travels on the same B7-B0 lane, a single word on B15-B0, and a quad on B31-B0. In a straight bus, however, a single byte can travel on either the B7-B0 lane or the B15-B8 lane, depending on whether it must be placed at an odd or at an even address. A bus is word-justified when only words are justified, but a byte within a word is not justified.

Justified buses include the Multibus,[12] which is byte-justified and little-endian; the S-100 bus,[13] which is byte-justified and big-endian; and the 32-bit Versabus/VME bus,[14,15] which is word-justified and big-endian. Note that all these standards specify a storage format in memory. Justification does not imply it—a justified bus is in principle no different from a straight bus with a multiplexed data path. The same observations as for the straight bus hold.

Justification requires that a single byte be recognized as such and be routed to the correct lane by a byte switch. This requirement is recognized in the Multibus, S-100 bus, and Versabus (Figure 11). Justification assumes that the processor indicates the width of its data; as we have seen above, this is not always the case. Processors such as the TI 9900 and the LSI-11 cannot be integrated into a justified bus.

The control lines of a justified bus indicate two things: the start address and the size of the data. The availability of such information looks appealing in terms

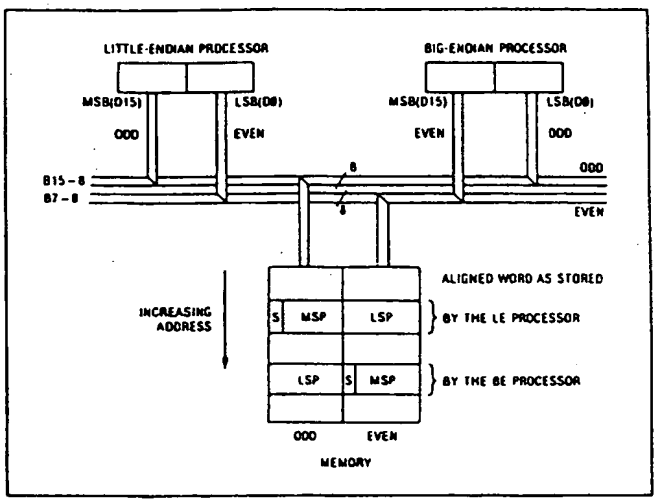

Figure 9. Straight (nonjustified) bus with a little-endian and a big-endian processor connected to it.

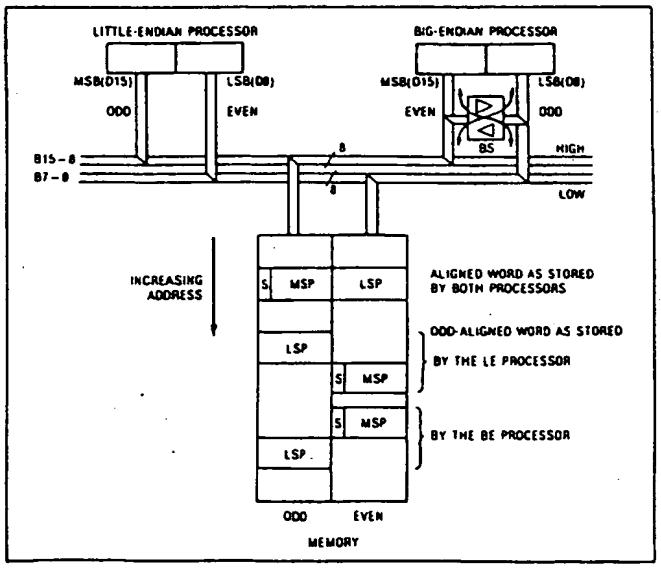

Figure 10. Trying to store words consistently over a straight bus.

of ensuring data format compatibility. One gives the start address of a data item and its size, and the bus interface is responsible for accessing the memory in the correct way. However, the data format will be defined only if one also indicates which half of the word is the MSP and which is the LSP. As shown in Figure 11, the justified bus respects the storage of bytes in memory, but handles words inconsistently.

The above-mentioned buses impose a data format in memory by coupling the odd/even with the high/low-byte indications. Strictly following these standards, one should be able to achieve a common data format by connecting the processors as shown in Figure 12. Alas, as was the case with straight buses, this method does not allow consistent storage of odd-aligned words. And here again one cannot rely on the byte/word indication of the processor. For these reasons, great care must be taken when interfacing a processor with a justified bus of the opposite data format.

There is potential trouble, for example, in connecting an 8086 or an NS 16000 (little-endians, pseudo-byte-aligned) to an S-100 bus (big-endian, jusitified), as shown in Figure 12. While aligned words will be stored in the correct big-endian format, words which begin at an odd address (and which are transmitted in two bytes) will be stored the little-endian way, because the interface is unable to distinguish the transfer of two halves of a word from the transfer of two single bytes. This should not greatly affect a big-endian processor like the MC68000 or the Z8000, since these processors cannot read odd-aligned words anyway. But the programmer should not access a word byte-wise, since the program will work differently when accessing local memory or global memory or when accessing an odd- or an even-aligned word.

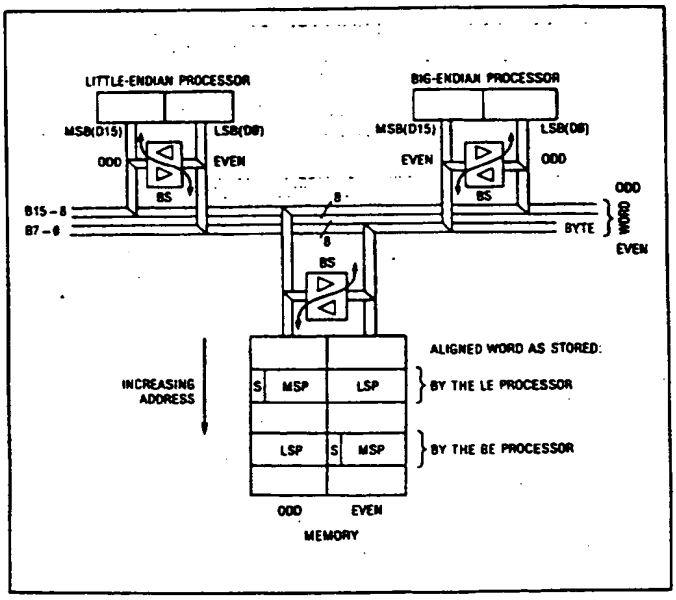

Figure 11. Interfacing a little-endian and a big-endian to a justified bus.

Why justify a bus?

Justification does not guarantee data format consistency—on the contrary, some processor types are ruled out when a bus is justified. So why are buses justified?

Justification allows communication between bus s of different physical width. For instance, the Multibus is justified to let 16-bit processors communicate with 8-bit peripherals. Justification requires a certain overhead in order to route data to the correct place, since the processor bus is always straight. This overhead consists of a single byte switch in a 16-bit system, but four are required in a 32-bit system. Figure 1 shows the logic required to route data for processors of different data width in a little-endian system. Let us compare this configuration with the same configuration on a straight bus (Figure 2). Here, each module must have access to all byte lanes. This puts the overhead of interfacing on the smaller systems. In a justified bus, transfers are always optimized for the participant with the smallest data path, e.g., for 8-bit devices in a byte-justified bus. The burden of compatibility is put on modules with wider data paths, e.g., on 16-bit modules in the Multibus.

In a mixed system comprising modules of different data widths, there are two reasons why one should introduce justification:

- To minimize cost. Since the majority of modules are of small width, overall cost can be reduced by burdening compatibility on the widest modules. This was the case at the time the Multibus was introduced.
- To accommodate smaller modules that do not have access to the whole width of the data path. The Versabus/VME bus is word-justified, since 16-bit systems do not have access to the full width of the 32-bit data path. (The higher data lines D16-D31 are on the second, facultative connector.)

Justification introduces some additional c n-straints. For instance, it obliges large-width modules to always speak in the language of small-width modules when dealing with them. A 16-bit module in a Multibus system must always access an 8-bit module by bytes. A smarter interface could, of course, automatically split a 16-bit word into two bytes when accessing an 8-bit device, but to do so it would have to receive a reply status from the accessed module telling it whether it was talking with an 8-bit or a 16-bit device.

Another constraint is that in a justified bus all transfers must be fully aligned. Doing otherwise would require that the interface be capable of swapping data. For instance, both halves of a word would have to be swapped if the word were accessed at an odd location (see Figure 7 in the main text of this article). This would cost four byte switches in a 16-bit system, and 16 byte switches in a 32-bit system. Fortunately, among processors today none send odd-aligned words; they use pseudo-byte alignment instead.

Justification has been introduced to accommodate small-width systems. We can categorize justified systems as follows:

- byte-justified = 8-bit optimized,
- word-justified = 16-bit optimized, and
- quad-justified = 32-bit optimized.

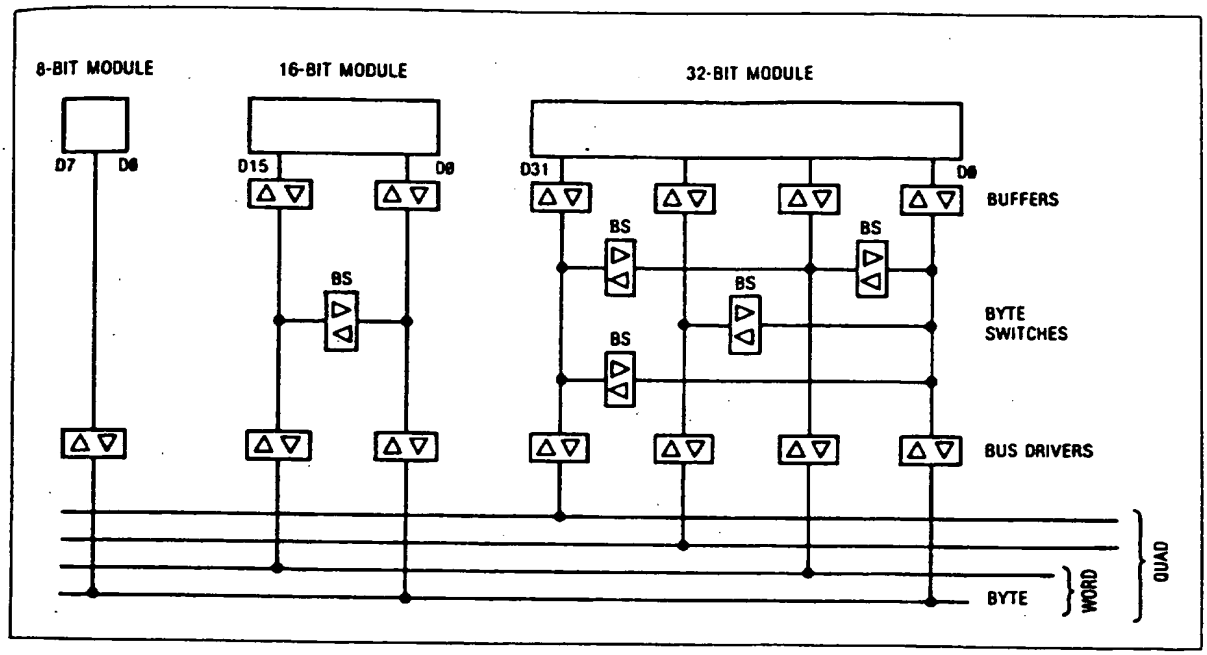

Figure 1. Interfacing 8-, 16-, and 32-bit modules to a justified little-endian bus.

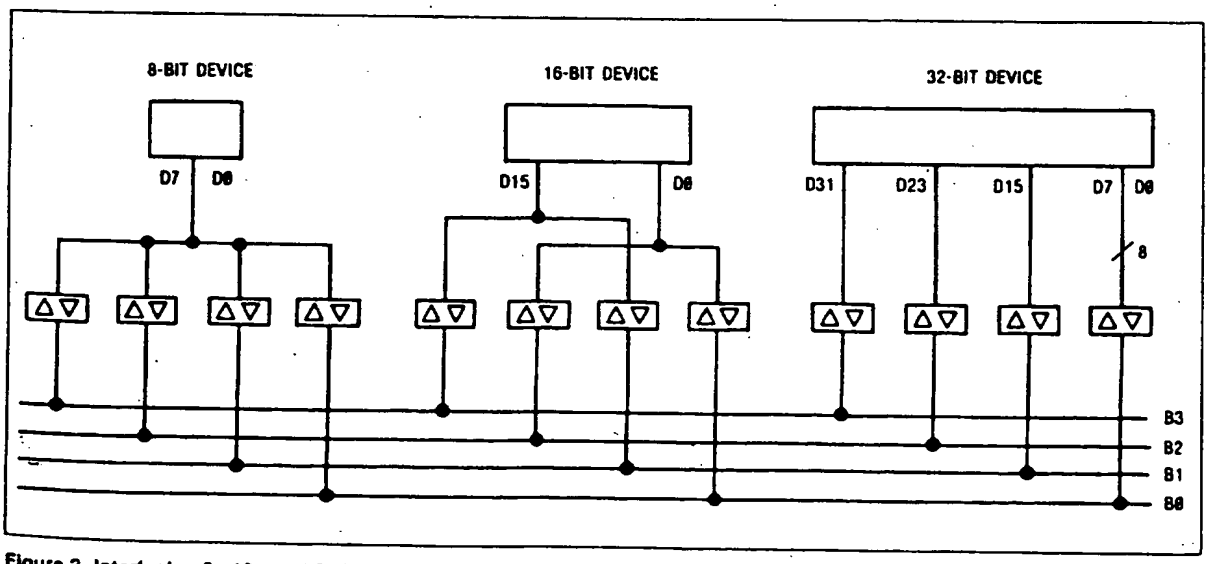

Figure 2. Interfacing 8-, 16-, and 32-bit modules to a straight little-endian bus.

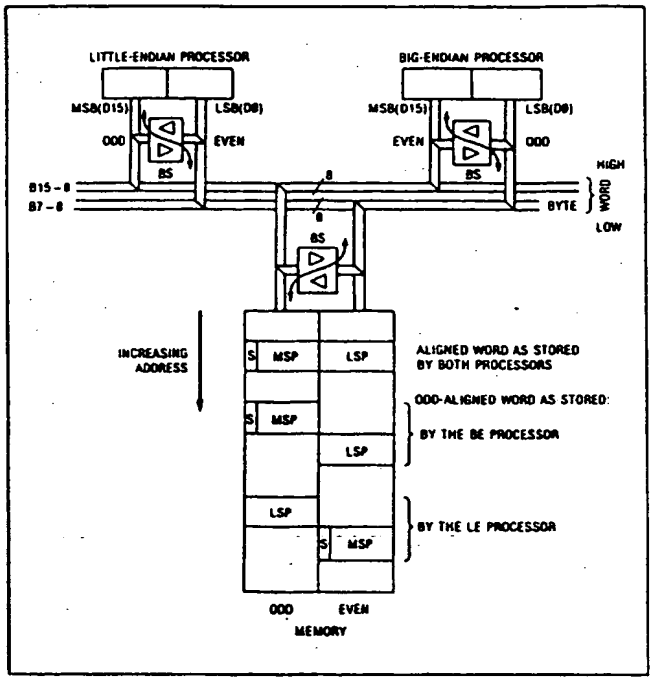

Figure 12. Trying to store words and bytes consistently over a justified bus.

On the other hand, it is easier to put an MC68000 or a Z8000 (big-endian) on a Multibus (justified little-endian), since these processors are fully aligned. The byte/word indication can then be used to indicate a single byte or word transfer, and the byte switch of Figure 12 will work. The Multibus can be made transparent to the processor, but the programmer must always access a word word-wise, not byte-wise.

The justified standard buses mentioned above are overspecified. To interface a processor of one type to a bus of the opposite type, the designer must violate the standard. In general, he should always try to have the bytes at the correct place (Figure 11), and he can only achieve consistent storage for words in a few rare cases (Figure 12).

We have seen that there is no common data format shared by the processors we have studied, except for the byte. Data format translation can be done by *software*. However, this translation can be quite time-consuming. The ideal alternative would be to let the *hardware* of the processor/bus interface do not only the physical adaptation, but also the data format translation. This is what has been attempted in the configurations shown in Figures 9 through 12.

To do data format translation, the interface must know which type of data is being transmitted. Unfortunately, no current processor indicates this. A logic analyzer connected to the processor bus is unable to

Should a bus be optimized for 16-bit or 32-bit transfers?

There is still no integrated 32-bit processor with a 32-bit-wide data path. The data path of the processors that claim to be 32-bit machines is still 16 bits wide, and such processors can just as well be called 64-bit machines, since their arithmetic unit can manage floating-point numbers of that size. So we will classify processors according to the width of their data path, and not according to the width of their internal registers. In our terminology the iAPX 432 is a 16-bit device.

But processors with 32-bit-wide data paths are bound to come. The interesting question is how their data paths will be organized and how future 32-bit system buses will look.

How advantageous is a 32-bit processor? It would seem that the throughput of a 32-bit processor would be twice that of a 16-bit processor, since it executes transfers on data twice the size of 16-bit data. In reality, this is only true if all transfers can be made 32 bits wide. This is not always possible. In a 32-bit system like the VAX, a small percentage of transfers is done on bytes (characters) and not on 32-bit entities. This percentage is application-dependent.

It is interesting to note that this problem most affects one application for which 32-bit systems are currently being developed—graphics processors. The screen is normally accessed as a RAM which is 16 bits wide (for color and symbols) and not 32 bits wide. The only operation in which 32-bit operations are interesting is copying one portion of a screen to another. Most other operations involve only one pixel or character at a time. This bus subutilization is, however, secondary. What most affects throughput is whether the 32-bit processor is byte-aligned or not. If the 32-bit processor is quad-aligned, i.e., if each quad is bound to a byte address divisible by 4, then the throughput will be effectively doubled. But communication is difficult in 16-bit systems, which are not constrained to operations on quad boundaries. It is especially difficult to make programs written for 16-bit processors compatible with a 32-bit machine.

On the other hand, if the 32-bit processor is byte-aligned, i.e., if any quad can begin on any byte address, then only one-fourth of all transfers will use the full bandwidth of the bus. Quads which are not quad-aligned will be fetched or written in two successive bus cycles, and they will account for three-fourths of all quad transfers (one-half of all word transfers, also). This is the case with the VAX, for example. The situation can be improved by processors that use a prefetch for instructions and (sometimes) variables. With such processors, only the variables—which account for about half of all transfers—will be affected by alignment.

Because of these considerations, we expect a 32-bit system to be only about 50 percent—not 100 percent—faster than a 16-bit system.

How should a mixed 16/32-bit system be optimized? There are two basic options:

- optimize buses for 16-bit-wide transfers, and
- optimize buses for 32-bit-wide transfers.

Both options are found in system buses. The Versabus, for instance, has 16-bit optimization, while the Fastbus[1] has 32-bit optimization.

16-bit optimization. The first option means that buses are optimized for 16-bit processors. A processor with a 16-bit-wide data path interfaces only to a 16-bit bus (Figure 1). The 32-bit bus is then word-justified, which has the nice side effect that a bus subset with less pins can be made for 16-bit processors.

The burden of interfacing is put on the 32-bit modules. Every 32-bit module must have a word switch in the form of two additional 8-bit buffers. These buffers can be efficiently implemented within the processor itself with no cost or delay penalty. Thirty-two-bit-wide memories must also have a word switch, although integrating that switch will be difficult. The total delay introduced by these additional buffers is normally negligible (about 30 nanoseconds).

An obvious disadvantage of this scheme is that a 32-bit processor can only communicate with a 16-bit memory by accessing it word-wise; i.e., it must know in advance whether it communicates with a 16- or a 32-bit memory. The communication protocol must ensure that the two participants in a data transfer always communicate on the level of the smallest data width.

32-bit optimization. In a 32-bit optimized system, the data path on the system bus is always 32 bits wide. Sixteen-bit systems that interface to this bus must have access to all 32 lines (Figure 2). Sixteen-bit devices are penalized by 16 additional bus drivers.

All memories ought to be 32 bits wide in a 32-bit optimized system. A 16-bit memory makes little sense,

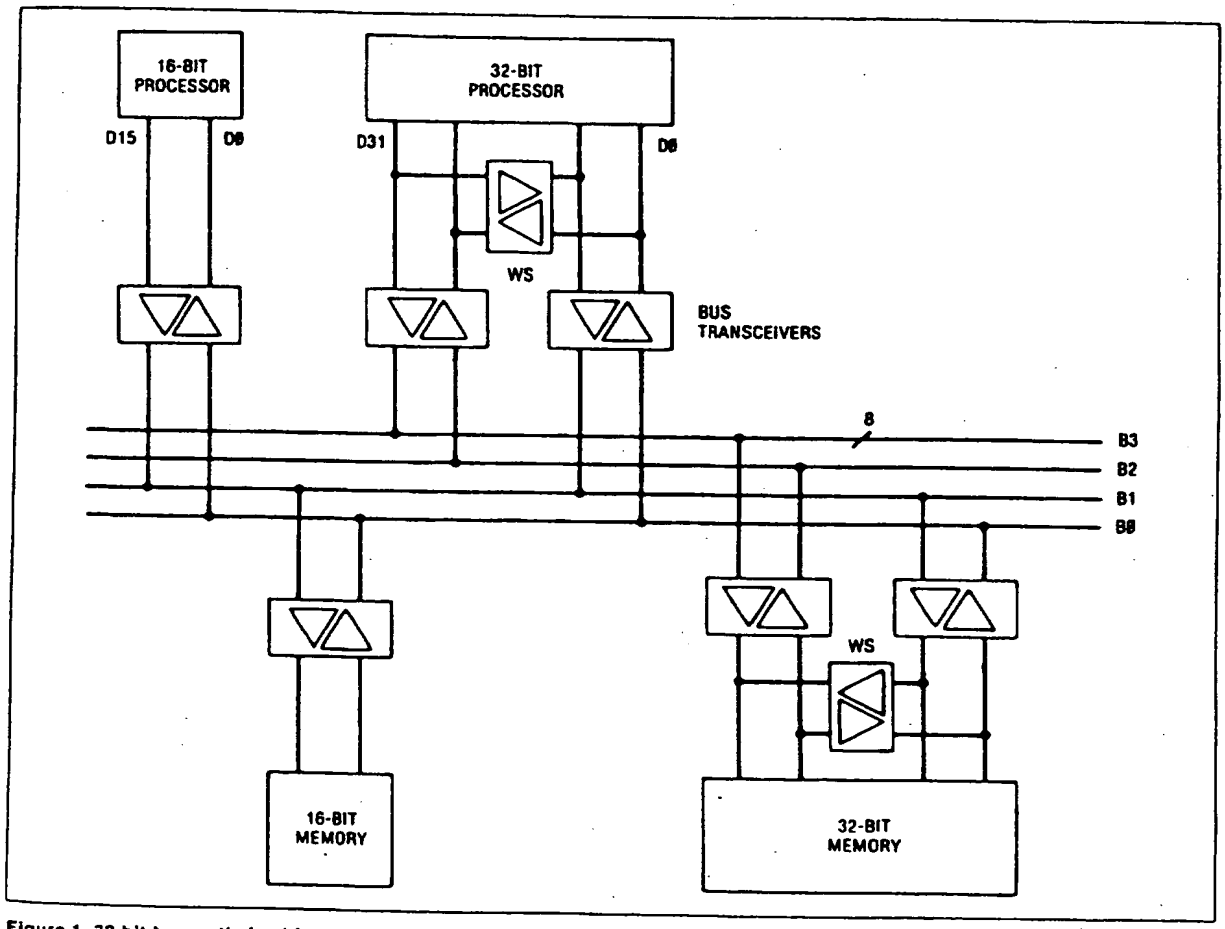

Figure 1. 32-bit bus optimized for 16-bit processors, with 16- and 32-bit devices attached.

but it may be required for operations such as accessing internal registers. However, a processor needs no knowledge of the memory's width, since a 16-bit memory will appear to the bus to be a (slow) 32-bit memory.

There is no time delay involved in a 32-bit optimized system. This solution requires additional power drivers, however. The 32-bit optimization also prevents the building of a 16-bit bus subset, which may be either an annoyance or a blessing, according to one's system philosophy.

Comparison. The 16- and 32-bit optimizations have about the same number of advantages and disadvantages (Table 1). We assume that future 32-bit processors will be tailored for high throughput but will nevertheless retain 16-bit compatibility to ease their introduction and lower system costs. For high throughput, a processor should be able to communicate without overhead with a 32-bit memory. To do this, a 32-bit processor must be able to independently steer each byte lane, either with a strobe (or a mask bit) per lane, or with a data length indication (byte/word/triplet/quad) plus the address at which the data should begin. (Each representation can be partially mapped onto the other, the second being somewhat more elegant.)

Table 1.
16-bit vs. 32-bit optimization for a 32-bit bus.

	16-BIT OPTIMIZED	32-BIT OPTIMIZED
TIME PENALTY WHEN TRANSMITTING		
A WORD	NONE	NONE
A QUAD TO A 16-BIT MODULE	TWICE THE TRANSFER TIME BECAUSE A QUAD MUST BE SPLIT INTO TWO WORDS*	TWICE THE ACCESS TIME FOR 16-BIT MEMORIES BECAUSE OF STORAGE IN CONTIGUOUS ADDRESSES ON THE SAME CHIP
A QUAD TO A 32-BIT MODULE	SAME AS ABOVE FOR A 16-BIT MODULE; NONE FOR A 32-BIT MODULE	NONE
LOGIC AMOUNT	TWO LOW-POWER 8-BIT BUFFERS FOR EACH 32-BIT MODULE	TWO 8-BIT BUS DRIVERS FOR EACH 16-BIT PROCESSOR (ONE IF ADDRESS/DATA MULTIPLEXED)
	ADDITIONAL LOGIC MAY BE REQUIRED FOR AUTOMATIC SPLITTING IN 32-BIT DEVICES	ADDITIONAL LOGIC REQUIRED FOR 16-BIT MODULES TO ASSEMBLE 32-BIT DATA
BUS SUBSET	16-BIT SUBSET POSSIBLE	NO 16-BIT SUBSET POSSIBLE
IMPACT ON DATA FORMAT	NONE	NONE

*Splitting is required anyway in 75 percent of the cases if the quads are not fully aligned.

In the interest of 16-bit compatibility, a 32-bit processor should probably have a 16-bit mode which uses word justification. Its internal shifter should allow the processor to perform justification at no cost. Although high throughput requires full alignment, 16-bit compatibility asks for byte alignment. Again, byte alignment is cheap to achieve internally, but programmers should be encouraged to use full alignment whenever possible.

To be integrated into a justified bus system, a 32-bit processor must issue data size information, even for READs. In the interest of compatibility and ease of program debugging, the processor should indicate on which type of data it is operating. This requirement goes further than just bus format compatibility and paves the way for a consistent system-wide object architecture.

Rationale for a 32-bit bus data format. There were several reasons why the EDISG chose a little-endian representation for its proposal[2] for the IEEE P896 backplane bus. One was that all 16-bit BE microprocessors (the MC68000, Z8000, and TI 9900) are word-aligned. Except for the TI 9900, which has no byte/word indication, these processors can be easily integrated into a little-endian world with a byte switch like that shown in Figures 10 and 12 in the main text (if some precautions in programming are taken), while LE processors are mostly pseudo-byte-aligned and cannot be easily fitted into a BE system. So in order to maximize manufacturer independence, a little-endian representation was recommended.

Another motive for the choice of a little-endian representation was that the Versabus and the VME bus are big-endian and therefore do not conform to the representation used in little-endian processors like the Intel 8086 or the NS 16000. Furthermore, the EDISG thought that 16-bit processors will dominate system design for some years to come, and therefore it recommended that the bus be 16-bit optimized.

References

1. US NIM Committee, "Fastbus Tentative Specification," US Dept. of Commerce, National Bureau of Standards, Washington, DC, Aug. 1981.
2. "P896 Draft 5.2," proposal of the P896 subgroup of the European Distributed Intelligence Study Group. (Available from the author at the address given in the biographical sketch at the end of this article.)

distinguish among the transfer of a 16-bit integer, a 16-bit address, four BCD digits, or the higher 16 bits of the mantissa of a floating-point number. The logic analyzer cannot even distinguish an instruction fetch from a data read in some processors like the LSI-11, unless it uses some tricky "manufacturer-reserved" lines.

The one indication a processor gives about the data it processes, the byte/word signal, can be used to convert the data format of words and bytes only when

- the processor issues and itself respects the byte/word indication (this is not the case for the TI 9900 and the LSI-11);
- the byte/word information indicates that the data transmitted are a single byte, and not the high or low part of a word (this rules out all pseudo-byte-aligned processors); and
- data are always read in the same format as they are written (this is left to the programmer's care).

Of all the processors we have discussed here, only the MC68000 and the Z8000 are suited for automatic format adaptation of bytes and words at the interface. With a little care in programming, the designer can quite easily integrate these processors into a little-endian world, and the processors can consistently store at least bytes and words, which are the most common data types.

All existing bus standards indirectly impose a data format for memory because they are overspecified. Hence, following the data type convention of one standard rules out processors of the opposite type. To overcome this limitation and standardize all data types, the interface would need to track the instruction flow of the processor and decide which type of data it is transmitting. The complexity of such an interface would approach that of the processor. Furthermore, such an interface would suppose that the processor itself has some knowledge of the data type. However, the processor has such knowledge only for hardware-defined types.

So until processors with standardized data types are available, the best thing is to leave the bus uncommitted and stick to the rule that bytes must be stored in the correct place. This can always be done, unless one uses a justified bus.

Automatic type conversion between processors having different data representations is restricted to very simple cases. It is possible to achieve partial compatibility of data items that are not wider than the bus width if one relies on the byte/word indication of some processors—as long as the data items are retrieved in the same format as they have been stored. This method is therefore not applicable to pseudo-byte-aligned processors and is currently restricted to the MC68000 and

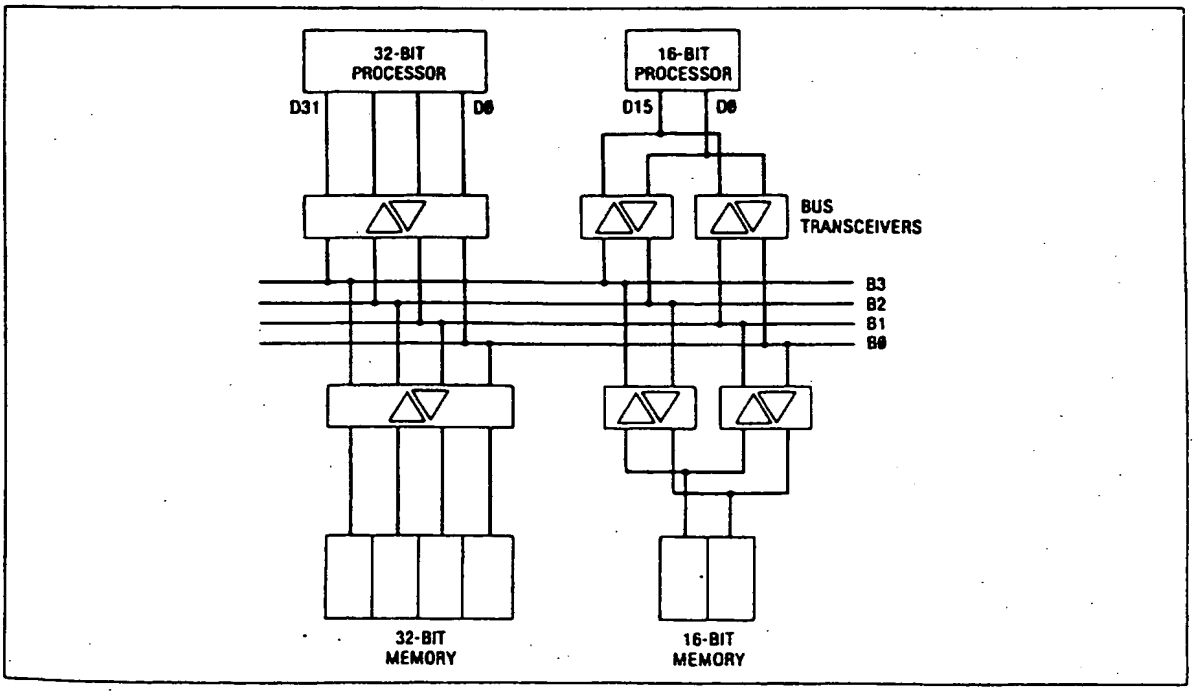

Figure 2. 32-bit bus optimized for 32-bit processors, with 16- and 32-bit devices attached.

the Z8000. The interface does not have sufficient knowledge to perform any other conversion, since the processor issues no information about the type of data it is manipulating and often has no such information.

Since an automatic format conversion at the interface between processor and bus is in most cases impractical, standard buses like the Multibus and S-100 bus should not impose a storage format for memory (e.g., by coupling the "low" byte with an "even" address). The only practical effect of imposing a data format is to favor the processor for which the bus was originally designed.

Although it is easier to only use homogeneous multiprocessors, the simple fact that data interchange media are standardized encourages people to build heterogeneous ones. Before it is too late, a common data interchange format should be standardized. I recommend that this standard be the little-endian format, since it is the natural way binary numbers are represented and since most existing processors can conform to it with little expense.

It is highly desirable that a processor in a multiprocessor system respect full alignment; i.e., an item of data must always be placed at an address that is a multiple of its size. Besides ensuring compatibility, full alignment speeds up execution on pseudo-byte-aligned processors. This is mostly a problem for compiler writers.

It is also desirable that data be read in the same format as it has been written. Buses should be justified only to connect buses of different physical widths. Justification does not guarantee any kind of data format com-

A proposal for a common data representation for multiprocessors

This proposal for a common data format for multiprocessors uses a little-endian rather than a big-endian representation. There is no compelling reason to choose one over the other, as long as whatever representation is chosen is consistent within itself (i.e., a processor should be LE or BE for all data and not LE for some and BE for other). One reason for choosing a little-endian format is its natural way of representing numbers. D7 is more significant than D0. Although we speak our numbers as BE, we add them in the LE way, beginning with the rightmost digit.

Communication chips are all little-endians. (However, disk controllers are often big-endians, and cyclic redundancy codes, as used in protocols like HDLC, are also big-endian.) An additional argument in favor of little-endian representation is that it is not difficult to integrate most existing BE processors into an LE system (at least for bytes and words, since no BE processors are pseudo-byte-aligned).

This proposal matches, to a certain extent, the representation used in Intel's 8087 and National's NS 16000. Except for the floating-point format, it is also compatible with DEC's LSI-11. The lowest significant bit of a datum has the lowest numbering, 0. Bits within a datum are numbered in decimal. The higher significant bit appears to the left, the lower significant bit to the right (e.g., AD <31...0> and not AD<0...31>). The following types of data are defined:

- INTEGER_8: An 8 bit integer, stored as a byte at any byte address. The least significant bit is number 0 and the most significant bit is number 7.
- CHAR_8: A character, stored in an 8-bit byte. ASCII format with even or no parity is recommended.
- INTEGER_16: A 16-bit integer, stored in two consecutive byte locations. The MSB with the sign is at the higher address, and the LSB in bit D0 is at the lowest address. This integer can represent an address <unsigned_integer> or a 2's-complement number <signed_integer>.
- INTEGER_32: A 32-bit integer, stored in four consecutive byte locations. The MSB with the sign is at the higher address, and the LSB in bit D0 is at the lowest address. This integer can represent an address <unsigned_integer> or a 2's-complement number <signed_integer>.
- INTEGER_64: A 64-bit integer, stored in four consecutive byte locations. The MSB with the sign is at the higher address, and the LSB in bit D0 is at the lowest address.
- FLOAT_32: A 32-bit floating-point number according to the proposed IEEE floating-point standard. The sign and most significant part of the exponent are at the higher address byte, and the LSB in bit D0 is at the lower address.
- FLOAT_64: A 64-bit floating-point number according to the proposed IEEE floating-point standard. The sign and most significant part of the exponent are at the higher address byte, and the LSB in bit D0 is at the lowest address.
- BCD_X: A string of *x* BCD numbers, each filling a nibble (four bits). The least significant nibble is at the lower address, and the least significant nibble within a byte is at D<3:0>.
- SET_X: A string of *x* Boolean bits. The first element of the set is in bit 0, which is at the lowest byte address.

Compound data types include

- ARRAYS: An array is stored with its lowest numbering element at the lowest address.
- RECORD: A record is stored with the first declared element at the lowest address.
- FILE: File elements are stored in the order a file is scanned, the first element being at the lowest address.
- TRANSMISSION ON A SERIAL MEDIUM: In a serial medium, the least significant bit is transmitted first. However, some arithmetic operations require the reverse convention in order to reduce the logic. Examples are CRC calculation and arbitration (comparison). In these cases, the breach in the convention should not appear at the next higher protocol layer.

patibility. On the contrary, it makes it difficult for a processor using one format to work on a bus which has the opposite format.

In future communication protocols, information about the type of the data should be transmitted along with the data itself in a standardized coded form.[16] This would greatly ease debugging and monitoring of distributed systems. In the meantime, it is good practice only to transmit 8-bit entities, like characters and 8-bit integers. The increased bandwidth of today's communication systems should lead designers to transmit data more and more in cleartext (ASCII) instead of compressed code. ■

Hubert D. Kirrmann is a research engineer at the Brown, Boveri & Company Research Center in Baden, Switzerland. He taught for several years at the District University of Bogota, Colombia, and he has worked as an R&D engineer in the field of transducers. He is currently involved in a multiprocessor project for process control and in computer bus standardization activities. His research interests include computer architecture and fault-tolerant systems.

A member of the IEEE, the ACM, and EWICS, Kirrmann received the MS degree in electrical engineering from the Swiss Federal Institute of Technology, Zürich, in 1970.

Kirrmann's address is Brown, Boveri & Company, Research Center, CH-5405 Baden-Dättwil, Switzerland.

References

1. A. A. Allison, "Status Report on the P896 Backplane Bus," *IEEE Micro*, Vol. 1, No. 1, Feb. 1981, pp. 67-82.
2. D. Cohen, "On Holy Wars and a Plea for Peace," *Computer*, Vol. 14, No. 10, Oct. 1981, pp. 48-54.
3. "Floating Point," chap. 10 in *Microcomputer Processor Handbook 1979/80*, Digital Equipment Corp., Maynard, MA, 1979.
4. *The 8086 Family User's Manual—Numeric Supplement*, Pub. No. S-21, Intel Corp., Santa Clara, CA, July 1979.
5. *16032 High-Performance Microprocessor*, National Semiconductor Corp., Santa Clara, CA, Apr. 1982.
6. *iAPX 43201/02 VLSI General Data Processor*, Intel Corp., Santa Clara, CA, 1981.
7. *MC68000 User's Guide*, chap. 2, Motorola, Inc., Austin, TX, 1979.
8. *AmZ8000 Microprocessor Specifications*, Advanced Micro Devices, Inc., Sunnyvale, CA, 1979, p. 17.
9. *9900 Family Systems Design and Data Book*, chap. 5, Texas Instruments, Inc., Dallas, TX, 1978.
10. *VAX-11/780 Architecture Handbook*, chap. 4, Digital Equipment Corp., Maynard, MA, 1977.
11. IEEE Task P754, "A Proposed Standard for Binary Floating-Point Arithmetic," *Computer*, Vol. 14, No. 3, Mar. 1981, pp. 51-62.
12. *Microcomputer System Bus (IEEE Standard 796-1983)*, Institute of Electrical and Electronics Engineers, Inc., New York, 1983. (This is the IEEE standard for the Multibus.)
13. *Interface Devices (IEEE Standard 696-1983)*, Institute of Electrical and Electronics Engineers, Inc., New York, 1983. (This is the IEEE standard for the S-100 bus.)
14. *Versabus Specification Manual*, Pub. No. M68KVBS(D4), Motorola, Inc., Phoenix, AZ, May 1981.
15. *VME Bus Specification Manual*, Pub. No. M68KVMEB(D1), Motorola, Inc., Phoenix, AZ, Oct. 1981.
16. M. Herlihy and B. Liskov, "Communicating Abstract Values in Messages," Tech. Memo. 200, Computation Structures Group, MIT, Cambridge, MA, Oct. 1980.